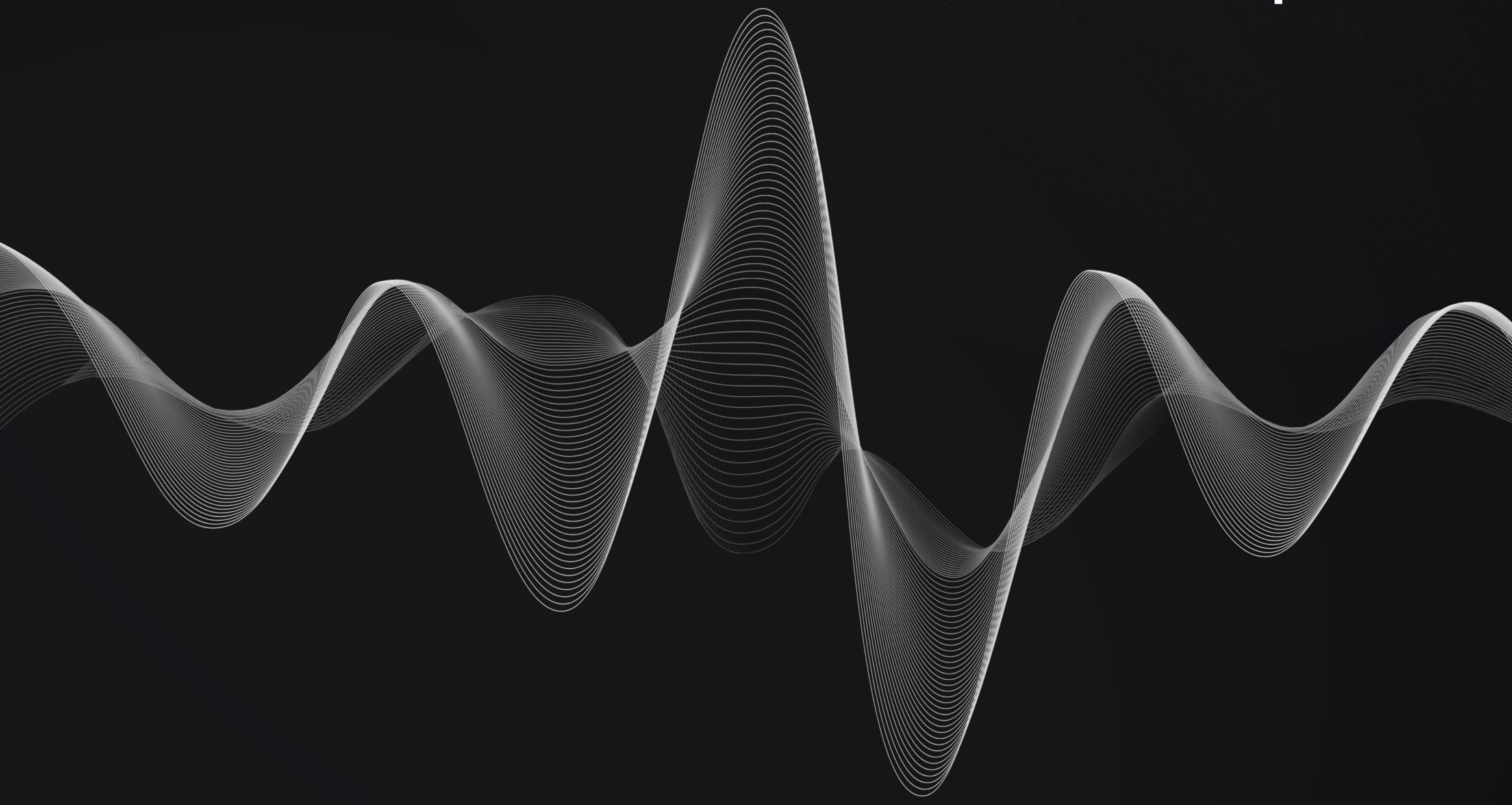
RESONANCE

DECENTRIC

Decentric IDO Platform Smart Contracts Audit Report



Blockchain, Emerging Technology, and Web2
CYBERSECURITY PRODUCT & SERVICE ADVISORY

Document Control

PUBLIC **FINAL**(v2.0)

Audit_Report_DCEN-IDO_FINAL_20

May 15, 2024	○	v0.1	João Simões: Initial draft
May 20, 2024	○	v0.2	João Simões: Added findings
May 21, 2024	●	v1.0	Charles Dray: Approved
Jun 11, 2024	○	v1.1	João Simões: Reviewed findings
Jun 26, 2024	○	v1.2	João Simões: Reviewed finding
Jun 26, 2024	●	v2.0	Charles Dray: Published

Points of Contact	Tom Sweeney	Decentric	tom@decentric.io
	Charles Dray	Resonance	charles@resonance.security
Testing Team	João Simões	Resonance	joao.simoes@resonance.security
	Ilan Abitbol	Resonance	ilan.abitbol@resonance.security
	Michał Bazyli	Resonance	michal.bazyli@resonance.security
	Michal Bajor	Resonance	michal.bajor@resonance.security

Copyright and Disclaimer

Contents

Executive Summary

Decentric contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts between May 7, 2024 and May 21, 2024. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated 2 engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted 10 days to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide Decentric with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.

System Overview

The Decentric IDO platform is a fair launchpad platform for projects to offer ERC20 assets directly to their early users via IDO. Users are eligible for rewards based on their on and off-chain reputation and activity within the protocol.

The system is composed of several components, the most important one being the launchpad where all the logic for handling and validating IDOs is. Additionally, the launchpad interacts with a lottery contract where tiered winners can claim ERC20 token rewards based on Chainlink VRF random numbers. The rewards can then be claimed or staked to get further access to the platform.

The protocol is meant to be deployed on Ethereum and EVM-compatible chains such as Arbitrum where DCEN tokens can be bridged back and forth via a LayerZero User Application with implemented antibot protection mechanisms.

Repository Coverage and Quality

Code	Tests	Documentation
9 / 10	8 / 10	9 / 10

Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development best practices and makes use of known patterns, standard libraries, and language guides. It is easily readable and uses the latest stable version of relevant components. Overall, **code quality is excellent**.

- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is 90%. Overall, **tests coverage and quality is good**.
- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is excellent**.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository: `REDACTED`
- Hash: de0c660c84d11fd3a25c0270399d61a68e177097

The following items are excluded:

- External and standard libraries
- Files pertaining to the deployment process
- Financial related attacks

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak **-** Inconsequential or minimal damage or loss
2. Medium **-** Temporary or partial damage or loss
3. Strong **-** Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely **-** Requires substantial knowledge or effort or uncontrollable conditions
2. Likely **-** Requires technical knowledge or no special conditions
3. Very Likely **-** Requires trivial knowledge or effort or no conditions

			Likelihood	
		Very Likely	Likely	Unlikely
Impact	Strong	Critical	High	Medium
	Medium	High	Medium	Low
	Weak	Medium	Low	Info

Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- ▂ **"Quick Win"** Requires little work for a high impact on risk reduction.
- ▄ **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- █ **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01	🟪	Unfair Advantage Of Winning Lottery Dependant On Number Of Contestants	█	Resolved
RES-02	🟥	Colluding Actors May Influence Winning Lottery	▄	Resolved
RES-03	🟥	Possible Griefing On payForUserBonus()	▄	Resolved
RES-04	🟥	Unclaimable Auto-unlocked Tokens Before Cliff	▄	Resolved
RES-05	🟧	Possible Griefing On depositFor()	▄	Resolved
RES-06	🟧	Incorrect Logic Leads to Unclaimable Tokens	█	Resolved
RES-07	🟧	Possible To Have Unclaimable Bonus Allocation Leftovers	▄	Resolved
RES-08	🟧	Missing Validation Of minimumPaymentPercentage_	▄	Resolved
RES-09	🟧	Insufficient Validation Of Duplicate Contestants	▂	Resolved
RES-10	🟩	Missing Zero Address Validation Of lottery	▂	Resolved
RES-11	🟩	Missing Zero Value Validation Of latestLeaderboardSnapshot	▂	Resolved
RES-12	🟩	Missing Zero Address Validation Of idoToken	▂	Resolved
RES-13	🟩	Missing Zero Value Validation Of requestId	▂	Resolved
RES-14	🟩	Immutable subscriptionId	▄	Resolved

RES-15	Public Function Returns Incorrect Information		Resolved
RES-16	Incorrect Validation Blocks User From Updating Stake Duration To Reasonable Values		Resolved
RES-17	Integer Underflow		Resolved

Unfair Advantage Of Winning Lottery Dependant On Number Of Contestants

Critical **RES-DCEN-IDO01** Business Logic **Resolved**

Code Section

- `src/lottery/Lottery.sol#L48-L63`
- `src/lottery/Lottery.sol#L66-L77`

Description

The function `runLottery()` is used to make a predetermined amount of winners out of an array of contestants. After performing necessary validation checks, it uses prefetched random numbers from Chainlink VRF to determine whether each individual contestant is a winner of the lottery for the specific tier. In essence, the function performs two important loops:

- On the first loop iterating through the contestants array, the function `calculateResult()` is called multiple times for the array's length. This function randomly determines whether the contestant is a winner. The variable `numberOfWinnersChosen` is incremented every time a winner is selected.
- On the second loop, the array of contestants is traversed in reverse order from the first loop. This loop only occurs if there were not enough winners selected on the first loop. Instead of calling the function `calculateResult()` again, contestants are automatically assigned a win if they were not winners yet. This guarantees that the function `runLottery()` will always yield the necessary amount of winners in order to return successfully

While not immediately perceivable, the previous logic introduces two major flaws that hamper the fairness of the lottery process:

- The result of the call to the function `calculateResult()` on the first loop is random and is an independent probabilistic event. This means that with an array of contestants of the same size as the number of expected winners, it is only natural that not all contestants will be selected as winners on the first loop. However, on the opposite side, if the array of contestants is very large in comparison to the number of expected winners, only the initial portion of the array will most likely yield winners, since once the expected number of winners is reached, the loop will break. Added to the fact that the addresses of contestants are numerically sorted, contestants with lower addresses will have a higher change of winning than their peers with higher addresses.
- Circling back to the scenario where the array of contestants is nearly the same size as the number of expected winners, when not enough winners are chosen on the first loop, the second loop will not calculate winners based on randomness. It will simply assign winners to contestants who are not winners yet, and it does so in reverse order. This means that higher addresses are more likely to simply win the lottery where lower addresses had to go through the random lottery process.

Essentially, whenever the size of the array of contestants is smaller, more winning probability is given to higher addresses, whereas whenever the size of the array of contestants is bigger, lower addresses have more probability of winning.

It should also be noted that, users may somewhat influence their own addresses prior to be assigned as contestants, e.g. keep creating new EOAs or contract addresses to serve their specific purpose as contestants.

Recommendation

It is recommended to review the logic of the lottery process and its randomness and fairness across all contestants.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Colluding Actors May Influence Winning Lottery

High **RES-DCEN-IDO02** Timestamp Dependence **Resolved**

Code Section

- `src/lottery/Lottery.sol#L95-L106`

Description

The function `calculateResult()` is a helper function that computes whether an individual contestant should be a winner or a loser on the lottery process. This function uses a random number prefetched from Chainlink VRF and encodes it into a `keccak256` hash along with other input parameters. One of these parameters is `block.timestamp`, which can be manipulated by either, the owner or manager of the protocol that is able to call the `runLottery()` function whenever he wants, or miners that validate the transaction.

Malicious contestants may collude with the owner or manager of the protocol, or miners, to fix the lottery process and indirectly assign themselves as winners.

It should also be noted that, while not as easy, other parameters such as `index`, `contestant` and `IDO_ID` may also be indirectly manipulated by malicious actors.

Recommendation

It is recommended to remove the dependency on the `block.timestamp` variable used to generate the `keccak256` hash.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Possible Griefing On payForUserBonus()

High **RES-DCEN-IDO03** Transaction Ordering **Resolved**

Code Section

- `src/payment/IDOPayment.sol#L189`

Description

The function `payForUserBonus()` can be called by users to pay for bonus allocation and further claim the respective allocation in reward tokens. This function allows users to pay on behalf of other users. While this functionality is relevant to support account abstraction mechanisms, it should be implemented with care.

The function performs a validation that checks whether the specific user has already paid for allocation, and if they have, it reverts. Since there are `maxPayment` and `minPaymentPercentage` variables that limit the input parameter `paymentAmount`, it is possible for malicious actors to frontrun transactions that call this function and grief legitimate users of paying for their allocation and receive reward tokens.

While most of the times, it is expected legitimate users make use of the `maxPayment` available to get the maximum amount of rewards, malicious users may perform the same transaction first and on their behalf with minimal cost. The impact of this vulnerable scenario directly depends on the cost of the exploit and the relevant variables `maxPayment` and `minPaymentPercentage`.

Recommendation

It is recommended to allow users to be able to pay for the allocation multiple times until the `maxPayment` value is reached.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Unclaimable Auto-unlocked Tokens Before Cliff

High **RES-DCEN-IDO04** Data Validation **Resolved**

Code Section

- `src/distribution/DistributionCalculator.sol#L31`

Description

The function `calculateClaimAmount()` performs vesting calculations on ERC20 tokens and returns the claimable amount to distributor contracts that can be claimed by users. An auto-unlock feature is also implemented that lets users claim tokens before the vesting process, and the amount of tokens that can be claimed at the start is controlled by the variable `percentageAvailableForClaim`.

The function performs a validation that checks if `percentageAvailableForClaim` is 100% and allows users to claim the auto-unlocked tokens. However, if `percentageAvailableForClaim` is anything different than 100%, the auto-unlocked tokens cannot be called on the start but only after the cliff period has ended.

Recommendation

It is recommended to adjust the validation being made on `block.timestamp` and allow users to be able to claim auto-unlocked tokens right at the `start` period, for every value of `percentageAvailableForClaim`.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Possible Griefing On depositFor()

Medium **RES-DCEN-IDO05** Transaction Ordering **Resolved**

Code Section

- `src/staking/Staking.sol#L108-L132`

Description

The function `depositFor()` is used to perform a lock of ERC20 tokens on behalf of a user that can be unstaked and withdrawn at a later date. While this functionality is relevant to support account abstraction mechanisms, it should be implemented with care.

The function also implements two protection mechanisms, the cancelation of the withdrawal request and the reset of the staking duration of the total amount of tokens.

Combining the previous two facts about the functionalities of this function, it enables the possibility for malicious actors to frontrun legitimate transactions that call this function and grief users on two possible accounts:

- Legitimate user stakes tokens. Stake duration has elapsed and legitimate user calls the function `requestWithdrawal()`. During the cooldown period, a malicious user deposits a minimal amount of tokens using `depositFor()`, causing the withdraw request to be cancelled. The malicious user may do this multiple times.
- Legitimate user stakes tokens. Stake duration is almost at the end or in withdraw cooldown period when a malicious user deposits a minimal amount of tokens using `depositFor()` with a duration equal to `maximumStakingDuration`, causing the user to have to wait further more to withdraw their tokens. The malicious user may do this multiple times.

While most of the times, it is expected legitimate users make use of the `minimumStakingDuration` to stake the tokens, malicious users may perform the same transaction first and on their behalf with minimal cost. The impact of this vulnerable scenario directly depends on the cost of the exploit and the relevant variables `minimumStakeAmount` and `maximumStakingDuration`.

Recommendation

It is recommended to adjust the functionality that allows for the cancellation of the withdraw request and possibly implement it in a new function. It is also recommended to perform validation checks to better accommodate the scenario where a user wishes to increase his stake of tokens, and the rest of the staking duration.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Incorrect Logic Leads to Unclaimable Tokens

Medium **RES-DCEN-IDO06** Business Logic **Resolved**

Code Section

- `src/distribution/DistributionCalculator.sol#L55`

Description

The function `calculateClaimAmount()` implements logic that hinders legitimate users from claiming tokens on a specific scenario. While it allows users to claim auto-unlocked and already vested tokens, in the case where the amount of leftover tokens to claim is less than what was already claimed minus the amount of auto-unlocked tokens, the function call reverts. This is due to calculations over the `adjustedClaimed` variable that will yield an underflow condition on the return statement of the function.

As a Proof of Concept example, suppose:

- `allocation` = 100 tokens
- `percentageAvailableForClaim` = 50 %
- `claimed` = 75 tokens

Then:

- `vesting` will be 50 tokens
- `unlocked` will be 12.5 tokens
- `adjustedClaimed` will be 25 tokens
- `unlocked - adjustedClaimed = -12.5` (Underflow)

In order to be able to retrieve the leftover tokens, the user will have to wait for the end of the vesting to claim all tokens, and can no longer claim tokens as the vesting process matures over time.

Recommendation

It is recommended to rectify the calculation of the `adjustedClaimed` variable and the return statement of the function in order to account for the vulnerable scenario.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Possible To Have Unclaimable Bonus Allocation Leftovers

Medium **RES-DCEN-IDO07** Business Logic **Resolved**

Code Section

- `src/payment/IDOPayment.sol#L195`

Description

The function `payForUserBonus()` performs a check that validates whether the provided `paymentAmount` is at least the `minPaymentPercentage` allowed. Furthermore, it does not allow the same user to pay for bonus allocation more than once. This effectively means that a user that provides incorrect input parameters by mistake may not rectify and increase their payment, losing on potential bonus rewards.

Recommendation

It is recommended to implement functionality to allow users to increase their payment for bonus allocation, or fix the amount that should be paid by every user.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Missing Validation Of minimumPaymentPercentage_

Medium **RES-DCEN-IDO08** Data Validation **Resolved**

Code Section

- `src/payment/IDOPayment.sol#L55-L57`
- `src/payment/IDOPayment.sol#L60-L62`
- `src/payment/IDOPayment.sol#L65-L67`
- `src/payment/IDOPayment.sol#L70-L72`

Description

The functions `updateMinimumPaymentPercentage()`, `updateMinimumPaymentPercentageForIDO()`, `updateMinimumBonusPaymentPercentage()`, and `updateMinimumBonusPaymentPercentageForIDO()` do not perform the same validations of the input parameter `minimumPaymentPercentage_` as the constructor of the smart contract does, leaving the possibility of undefined behaviour across the protocol, should these variable be set incorrectly.

Recommendation

It is recommended to maintain consistency regarding the code logic and relevant validations on getters and setters of the same variable across the smart contracts and the protocol.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Insufficient Validation Of Duplicate Contestants

Medium **RES-DCEN-IDO09** Data Validation **Resolved**

Code Section

- `src/lottery/Lottery.sol#L53`

Description

The function `runLottery()` does not properly validate duplicate entries on the `contestants` array. While the array is assumed to be sorted numerically in ascending order, the validation does not check for equality with the previous entry.

Recommendation

It is recommended to adjust the validation to revert the transaction if the current address entry on the loop is less than or equal to the previous one.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Missing Zero Address Validation Of lottery

Low

RES-DCEN-IDO10 Data Validation **Resolved**

Code Section

- `src/launchpad/Launchpad.sol`

Description

Throughout the Launchpad smart contract the `lottery` state variable is used to perform external calls. However, this variable is not validated against the Zero Address, allowing for undefined behavior within the protocol.

Recommendation

It is recommended to perform a validation against the Zero Address to ensure external calls are handled properly and successfully.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Missing Zero Value Validation Of latestLeaderboardSnapshot

Low

RES-DCEN-IDO11 Data Validation **Resolved**

Code Section

- `src/launchpad/Launchpad.sol#L81-L84`
- `src/launchpad/Launchpad.sol#L135-L138`

Description

The functions `updateLeaderboardSnapshot()` and `resyncLeaderboardSnapshotWithIDO()` do not validate the variable `latestLeaderboardSnapshot` against the Zero Value, allowing for undefined behavior within the protocol.

It should be noted that this fact becomes specially relevant since these functions can be called after the successful execution of the function `runLottery()`, where further validation checks will be made to this variable.

Recommendation

It is recommended to perform a validation against the Zero Value to ensure all interactions with the variable return proper and successful results.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Missing Zero Address Validation Of idoToken

Low **RES-DCEN-IDO12** Data Validation **Resolved**

Code Section

- `src/Common.sol#L42-L51`

Description

The function `validateVestingTerms()` does not validate the variable `idoToken` against the Zero Address, allowing for undefined behavior within the protocol.

It should be noted that this fact becomes specially relevant since this variable is only interacted with much further down the code workflow, when it becomes very difficult to justify a genuine update of vesting terms.

Recommendation

It is recommended to perform a validation against the Zero Address to ensure external calls are handled properly and successfully.

Status

The issue has been fixed in 6c45f09afa4a83fd893cd7b317a6e99208cd532b.

Missing Zero Value Validation Of requestId

Low

RES-DCEN-ID013 Data Validation **Resolved**

Code Section

- `src/lottery/VRF.sol#L70-L76`

Description

The function `getRequestStatus()` does not validate the variable `requestId` against the Zero Value, allowing for incorrect information to be presented by the protocol. Even though a `requestId` of `0` would most likely be invalid, the protocol can still confirm it exists even when it does not.

Recommendation

It is recommended to perform a validation against the Zero Value to ensure all interactions with the variable return proper and successful results.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Immutable subscriptionId

Low

RES-DCEN-IDO14 Denial of Service **Resolved**

Code Section

- `src/lottery/VRF.sol#L42`

Description

The variable `subscriptionId` is not capable of being modified after being initialized in the constructor. Since this variable relates to Chainlink's subscription to request random numbers, if the subscription account is compromised, the protocol may become compromised as well.

Recommendation

It is recommended to implement code logic to be able to modify the `subscriptionId` variable. The ability to modify such variable should be access controlled.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Public Function Returns Incorrect Information

Low **RES-DCEN-IDO15** Data Validation **Resolved**

Code Section

- `src/launchpad/Launchpad.sol#L199`

Description

The function `isWinnerInTierInIDO()` does not properly validate whether a contestant is actually a winner in the specific case of providing `TierOne` as the `tier` input parameter, yielding a possibly incorrect result of `true`.

It should be noted that while this function is being used internally by the protocol to perform relevant checks, it still provides erroneous information should it be called directly by a legitimate user of the protocol.

Recommendation

It is recommended to properly distinguish in the code between public/external and private/internal functions and where each shall/can be used. It should also be considered to move the `TierOne` verification logic out of the public function.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Incorrect Validation Blocks User From Updating Stake Duration To Reasonable Values

Low

RES-DCEN-IDO16 Data Validation **Resolved**

Code Section

- `src/staking/Staking.sol#L177`

Description

The function `updateStakingDuration()` does not properly validate the input parameter `newStakingDuration`. Apart from other validations, the variable `newStakingDuration` is being compared to `stakingEndTimestamp`, which means that a duration is being incorrectly compared to a timestamp.

This fact results in extreme modifications to the duration of the token staking.

Recommendation

It is recommended to properly implement validations that compare variables of the same order of magnitude and applicability, i.e. either perform the necessary comparison between two timestamps or between two durations.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Integer Underflow

Info **RES-DCEN-IDO17** Arithmetic Issues **Resolved**

Code Section

- `src/lottery/Lottery.sol#L68`

Description

The function `runLottery()` does not properly protect against an integer underflow condition of the variable `uint256 i` on the second for loop. As the variable `i` iterates through the `contestants` array, it is decremented at the end of each iteration. On the last iteration, it will again attempt to decrement `i` below `0` which for a `uint256` variable is not possible, thus resulting in a integer underflow.

It should be noted that this scenario is only possible when not enough winners are selected, which would revert the transaction all the same.

Recommendation

It is recommended to adjust the loop condition to never try to decrement the indexer variable below `0` on a `uint256`, or change the indexer variable to a `int256`.

Status

The issue has been fixed in 9c9e80858ed08471de0a8e3b519bdcca5b822baa.

Proof of Concepts

RES-03 Possible Griefing On payForUserBonus()

Launchpad.t.sol (added lines):

```
function testExploitGriefingPayForUserBonus(
    uint128 randomNumOne,
    uint128 randomNumTwo,
    uint128 randomNumThree,
    uint128 randomNumFour,
    uint128 randomNumFive
) public {
    // Same as testSmallScaleDistributionOnBehalfOfUsers, see the end of the test
    uint256 IDO_ID = 1;

    // Create users for all tiers
    (
        address[] memory tierOne,
        address[] memory tierTwo,
        address[] memory tierThree,
        address[] memory tierFour,
        address[] memory bonus
    ) = createSmallUserSetForAllTiers(randomNumOne, randomNumTwo, randomNumThree,
↪   randomNumFour, randomNumFive);

    vm.prank(platform);
    launchpad.updateLeaderboardSnapshot(keccak256("Empty"));

    // Run the lottery
    Launchpad.IDO memory ido = runLotteryForAllTiers(IDO_ID, randomNumOne,
↪   randomNumTwo, randomNumThree, tierTwo, tierThree, tierFour);

    // Set the leaderboard snapshot
    (bytes32 leaderboardRoot, bytes32[] memory data) =
↪   generateTreeForWinners(IDO_ID, tierOne, tierTwo, tierThree, tierFour, bonus);
    vm.startPrank(platform);
    launchpad.updateLeaderboardSnapshot(leaderboardRoot);
    launchpad.resyncLeaderboardSnapshotWithIDO(IDO_ID);
    vm.stopPrank();

    // Test payments
    assert(launchpad.isOpenForPayment(IDO_ID) == false);
    vm.warp(ido.paymentTerms.paymentAllocationStartTime + 1 seconds);
    assert(launchpad.isOpenForPayment(IDO_ID) == true);

    uint256 nextIndex;
    (uint256 maxPayment,,) = launchpad.getMaxPaymentForTier(IDO_ID,
↪   Common.Tier.TierOne);
    for (uint256 i; i < tierOne.length; ++i) {
        makePaymentOnBehalf(IDO_ID, Common.Tier.TierOne, nextIndex + 1, maxPayment,
↪   tierOne[i], accountTwo, data);
```

```
        nextIndex += 1;
    }

    assertEq(PaymentERC20(paymentToken).balanceOf(address(payment.vault())),
↪   maxPayment);

    for (uint256 i; i < tierTwo.length; ++i) {
        if (lottery.isWinnerInTierInIDO(IDO_ID, Common.Tier.TierTwo, tierTwo[i])) {
            makePaymentOnBehalf(IDO_ID, Common.Tier.TierTwo, nextIndex + 1,
↪   maxPayment, tierTwo[i], accountTwo, data);
            nextIndex += 1;
        }
    }

    assertEq(PaymentERC20(paymentToken).balanceOf(address(payment.vault())),
↪   maxPayment * 11);

    for (uint256 i; i < tierThree.length; ++i) {
        if (lottery.isWinnerInTierInIDO(IDO_ID, Common.Tier.TierThree,
↪   tierThree[i])) {
            makePaymentOnBehalf(IDO_ID, Common.Tier.TierThree, nextIndex + 1,
↪   maxPayment, tierThree[i], accountTwo, data);
            nextIndex += 1;
        }
    }

    assertEq(PaymentERC20(paymentToken).balanceOf(address(payment.vault())),
↪   maxPayment * 31);

    for (uint256 i; i < tierFour.length; ++i) {
        if (lottery.isWinnerInTierInIDO(IDO_ID, Common.Tier.TierFour, tierFour[i]))
↪   {
            makePaymentOnBehalf(IDO_ID, Common.Tier.TierFour, nextIndex + 1,
↪   maxPayment, tierFour[i], accountTwo, data);
            nextIndex += 1;
        }
    }

    assertEq(PaymentERC20(paymentToken).balanceOf(address(payment.vault())),
↪   maxPayment * 111);

    assert(launchpad.isOpenForBonus(IDO_ID) == false);
    vm.warp(ido.paymentTerms.bonusStartTime + 1 seconds);
    assert(launchpad.isOpenForBonus(IDO_ID) == true);

    // Exploit here
    // Attacker frontruns with minimum payment 150 * 12% = 18
    makeBonusPaymentOnBehalf(IDO_ID, bonus[0], accountTwo, 18 ether);
    // Victim cannot claim bonus anymore
    // Will revert with InvalidOperation
    makeBonusPaymentOnBehalf(IDO_ID, bonus[0], accountTwo, 150 ether);
}
```

RES-04 Unclaimable Auto-unlocked Tokens Before Cliff

SimpleDistributor.t.sol (added lines):

```
function testExploitAutoUnlockBeforeCliff() public {
    uint256 time = block.timestamp;
    SimpleDistributor.WalletVestingInfo memory info =
↪   createUserWithPrimaryAllocationAutoUnlock90(207 ether, time);

    address[] memory users = new address[](1);
    users[0] = accountTwo;

    SimpleDistributor.WalletVestingInfo[] memory infos = new
↪   SimpleDistributor.WalletVestingInfo[](1);
    infos[0] = info;

    vm.prank(platform);
    distributor.updateVestingInfo(users, infos);

    // transfer tokens
    vm.prank(platform);
    decentricToken.transfer(address(distributor), 207 ether);

    // Commenting warp to simmulate claim before cliff
    // Should be able to get 90% AutoUnlocked
    //vm.warp(time + 90 minutes);

    vm.prank(accountTwo);
    distributor.claim();

    assertEq(decentricToken.balanceOf(accountTwo), 186.3 ether);
}

function createUserWithPrimaryAllocationAutoUnlock90(uint256 allocation, uint256
↪   fromTime) internal
view returns (SimpleDistributor.WalletVestingInfo memory) {
    SimpleDistributor.WalletVestingInfo memory info;

    SimpleDistributor.VestingTermsWithAllocation memory termsWithAllocation =
↪   SimpleDistributor.VestingTermsWithAllocation(
        allocation,
        Common.VestingTerms(
            address(decentricToken),
            90_000,
            uint64(fromTime + 15 minutes),
            uint64(fromTime + 45 minutes),
            uint64(fromTime + 15 minutes)
        )
    );

    info.primarySchedule = termsWithAllocation;

    return info;
}
```

RES-05 Possible Griefing On depositFor()

Staking.t.sol (added lines):

```
function testExploitGriefingStakeDuration(uint128 depositAmount) public {
    vm.assume(depositAmount >= minStakeAmount && depositAmount <= 1_000_000_000
↪   ether - minStakeAmount);
    uint128 stakingDuration = minStakingDuration;

    // Preparation -> Give balance to malicious actor
    vm.prank(platform);
    decentricToken.transfer(address(0x1), minStakeAmount);


    // Legitimate staking
    execute_deposit(platform, depositAmount, stakingDuration);

    assert(decentricToken.balanceOf(address(staking)) == depositAmount);

    (uint128 stakingEndTimestamp, uint128 amount) =
↪   staking.userStakeMetadata(platform);
    assert(amount == depositAmount);
    assert(stakingEndTimestamp == block.timestamp + stakingDuration);

    uint256 timeBefore = block.timestamp;
    // 10 days until able to withdraw
    vm.warp(block.timestamp + stakingDuration - 10 days);

    // Insert cheap illegitimate depositFor to grief victim
    execute_deposit_for(address(0x1), minStakeAmount,
↪   maxStakingDurationUpperBoundForTesting, platform);

    // Staking duration was reset and increased to max by malicious actor
    (stakingEndTimestamp, ) = staking.userStakeMetadata(platform);
    assert(stakingEndTimestamp == timeBefore + stakingDuration - 10 days +
↪   maxStakingDurationUpperBoundForTesting);
}

function testExploitGriefingWithdrawalRequest(uint128 depositAmount) public {
    vm.assume(depositAmount >= minStakeAmount && depositAmount <= 1_000_000_000
↪   ether - minStakeAmount);
    uint128 stakingDuration = minStakingDuration;

    // Preparation -> Give balance to malicious actor
    vm.prank(platform);
    decentricToken.transfer(address(0x1), minStakeAmount);


    // Legitimate staking
    execute_deposit(platform, depositAmount, stakingDuration);
    // Warp past the lock and request withdrawal
    vm.warp(block.timestamp + minStakingDuration + 1 hours);
    vm.prank(platform);
    staking.requestWithdrawal();

    uint256 userWithdrawalTimestamp = staking.userWithdrawalTimestamp(platform);
    assert(userWithdrawalTimestamp == block.timestamp + coolDownPeriod);

    // Warp almost past cool down
    vm.warp(block.timestamp + coolDownPeriod - 1 minutes);

    // Insert cheap illegitimate depositFor to grief victim
    execute_deposit_for(address(0x1), minStakeAmount,
↪   maxStakingDurationUpperBoundForTesting, platform);

    // Withdrawal request was reset
    userWithdrawalTimestamp = staking.userWithdrawalTimestamp(platform);
    assert(userWithdrawalTimestamp == 0);
}

function execute_deposit_for(
    address user,
    uint128 amount,
    uint128 duration,
    address recipient
) internal {
    vm.startPrank(user);
    decentricToken.approve(address(staking), amount);
    staking.depositFor(recipient, amount, duration);
    vm.stopPrank();
}
```

RES-06 Incorrect Logic Leads to Unclaimable Tokens

SimpleDistributor.t.sol (added lines):

```
function testExploitUnclaimableTokens() public {
    uint256 time = block.timestamp;
    SimpleDistributor.WalletVestingInfo memory info =
↪   createUserWithPrimaryAllocationAutoUnlock50(100 ether, time);

    address[] memory users = new address[](1);
    users[0] = accountTwo;

    SimpleDistributor.WalletVestingInfo[] memory infos = new
↪   SimpleDistributor.WalletVestingInfo[](1);
    infos[0] = info;

    vm.prank(platform);
    distributor.updateVestingInfo(users, infos);
    // transfer tokens
    vm.prank(platform);
    decentricToken.transfer(address(distributor), 100 ether);

    // Claim AutoUnlocked (50 tokens) + 30 minutes worth of vesting (25 tokens)
    vm.warp(time + 30 minutes);

    vm.prank(accountTwo);
    distributor.claim();

    assertEq(decentricToken.balanceOf(accountTwo), 75 ether);

    // Claim 15 more minutes worth of vesting (12.5 tokens)
    vm.warp(time + 45 minutes);

    vm.prank(accountTwo);
    distributor.claim();

    assertEq(decentricToken.balanceOf(accountTwo), 87.5 ether);
}

function createUserWithPrimaryAllocationAutoUnlock50(uint256 allocation, uint256
 ↪  fromTime) internal
view returns (SimpleDistributor.WalletVestingInfo memory) {
    SimpleDistributor.WalletVestingInfo memory info;

    SimpleDistributor.VestingTermsWithAllocation memory termsWithAllocation =
 ↪  SimpleDistributor.VestingTermsWithAllocation(
        allocation,
        Common.VestingTerms(
            address(decentricToken),
            50_000,
            uint64(fromTime),
            uint64(fromTime + 1 hours),
            uint64(fromTime)
        )
    );

    info.primarySchedule = termsWithAllocation;

    return info;
}
```

RES-07 Possible To Have Unclaimable Bonus Allocation Leftovers

Launchpad.t.sol (added lines):

```
function testExploitUnclaimableLeftovers(
    uint128 randomNumOne,
    uint128 randomNumTwo,
    uint128 randomNumThree,
    uint128 randomNumFour,
    uint128 randomNumFive
) public {
    // Same as testSmallScaleDistributionOnBehalfOfUsers, see the end of the test
    uint256 IDO_ID = 1;

    // Create users for all tiers
    (
        address[] memory tierOne,
        address[] memory tierTwo,
        address[] memory tierThree,
        address[] memory tierFour,
        address[] memory bonus
    ) = createSmallUserSetForAllTiers(randomNumOne, randomNumTwo, randomNumThree,
↪   randomNumFour, randomNumFive);

    vm.prank(platform);
    launchpad.updateLeaderboardSnapshot(keccak256("Empty"));

    // Run the lottery
    Launchpad.IDO memory ido = runLotteryForAllTiers(IDO_ID, randomNumOne,
↪   randomNumTwo, randomNumThree, tierTwo, tierThree, tierFour);

    // Set the leaderboard snapshot
    (bytes32 leaderboardRoot, bytes32[] memory data) =
↪   generateTreeForWinners(IDO_ID, tierOne, tierTwo, tierThree, tierFour, bonus);
    vm.startPrank(platform);
    launchpad.updateLeaderboardSnapshot(leaderboardRoot);
    launchpad.resyncLeaderboardSnapshotWithIDO(IDO_ID);
    vm.stopPrank();

    // Test payments
    assert(launchpad.isOpenForPayment(IDO_ID) == false);
    vm.warp(ido.paymentTerms.paymentAllocationStartTime + 1 seconds);
    assert(launchpad.isOpenForPayment(IDO_ID) == true);

    uint256 nextIndex;
    (uint256 maxPayment,,) = launchpad.getMaxPaymentForTier(IDO_ID,
↪   Common.Tier.TierOne);
    for (uint256 i; i < tierOne.length; ++i) {
        makePaymentOnBehalf(IDO_ID, Common.Tier.TierOne, nextIndex + 1, maxPayment,
↪   tierOne[i], accountTwo, data);
        nextIndex += 1;
    }

    assertEq(PaymentERC20(paymentToken).balanceOf(address(payment.vault())),
↪   maxPayment);

    for (uint256 i; i < tierTwo.length; ++i) {
        if (lottery.isWinnerInTierInIDO(IDO_ID, Common.Tier.TierTwo, tierTwo[i])) {
            makePaymentOnBehalf(IDO_ID, Common.Tier.TierTwo, nextIndex + 1,
↪   maxPayment, tierTwo[i], accountTwo, data);
            nextIndex += 1;
        }
    }

    assertEq(PaymentERC20(paymentToken).balanceOf(address(payment.vault())),
↪   maxPayment * 11);

    for (uint256 i; i < tierThree.length; ++i) {
        if (lottery.isWinnerInTierInIDO(IDO_ID, Common.Tier.TierThree,
↪   tierThree[i])) {
            makePaymentOnBehalf(IDO_ID, Common.Tier.TierThree, nextIndex + 1,
↪   maxPayment, tierThree[i], accountTwo, data);
            nextIndex += 1;
        }
    }

    assertEq(PaymentERC20(paymentToken).balanceOf(address(payment.vault())),
↪   maxPayment * 31);

    for (uint256 i; i < tierFour.length; ++i) {
        if (lottery.isWinnerInTierInIDO(IDO_ID, Common.Tier.TierFour, tierFour[i]))
↪   {
            makePaymentOnBehalf(IDO_ID, Common.Tier.TierFour, nextIndex + 1,
↪   maxPayment, tierFour[i], accountTwo, data);
            nextIndex += 1;
        }
    }

    assertEq(PaymentERC20(paymentToken).balanceOf(address(payment.vault())),
↪   maxPayment * 111);

    assert(launchpad.isOpenForBonus(IDO_ID) == false);
    vm.warp(ido.paymentTerms.bonusStartTime + 1 seconds);
    assert(launchpad.isOpenForBonus(IDO_ID) == true);

    // Exploit here
    (uint256 maxPaymentForTierBonus,,) = launchpad.getMaxPaymentForTier(IDO_ID,
↪   Common.Tier.Bonus);

    // User mistakenly forgets a 0 digit or inputs a wrong digit
    makeBonusPaymentOnBehalf(IDO_ID, bonus[0], accountTwo, maxPaymentForTierBonus -
↪   10 ether);

    // User tries to claim the remaining leftovers but is unable due to both:
    // - Having already claimed 140 ether (IDOPayment#L189), and
    // - minPaymentPercentage, equivalent to 18 ether, is greater than 10 ether
↪   (IDOPayment#L195)
    makeBonusPaymentOnBehalf(IDO_ID, bonus[0], accountTwo, 10 ether);
}
```

RES-09 Insufficient Validation Of Duplicate Contestants

Launchpad.t.sol (added lines):

```
function testExploitDuplicateContestants(
    uint128 randomNumOne,
    uint128 randomNumTwo,
    uint128 randomNumThree
) public {
    uint256 IDO_ID = 1;
    prepareLotteryForDefaultIDO(IDO_ID, randomNumOne, randomNumTwo, randomNumThree);

    // Create 150 contestants of which 90 will win
    address[] memory contestants = createContestants(randomNumTwo, 150);
    // First 10 addresses are all duplicates of address(0x1)
    for (uint256 i = 0; i < 10; i++) {
        contestants[i] = address(0x01);
    }

    // Run the lottery
    vm.prank(platform);
    launchpad.runLottery(IDO_ID, Common.Tier.TierTwo, contestants);
}
```

RES-16 Incorrect Validation Blocks User From Updating Stake Duration To Reasonable Values

Staking.t.sol (added lines):

```
function testExploitUpdateStakingDuration(uint128 depositAmount) public {
    // Set date as January 1st, 2024
    vm.warp(1704067200);

    vm.assume(depositAmount >= minStakeAmount && depositAmount <=
↪   decentricToken.balanceOf(platform));
    execute_deposit(platform, depositAmount, minStakingDuration);

    // stakingEndTimestampBefore is March 31st, 2024 (1711843200)
    (uint128 stakingEndTimestampBefore, ) = staking.userStakeMetadata(platform);
    assert(stakingEndTimestampBefore == (block.timestamp + (minStakingDuration)));

    // Warp past the lock
    vm.warp(block.timestamp + minStakingDuration + 1 hours);

    // Staking.sol#L177: if (newStakingDuration < userStake.stakingEndTimestamp) ->
↪   reverts
    // newStakingDuration: minStakingDuration * 2 = 90 days * 2 = 15552000
    // stakingEndTimestamp: 1711843200
    // These input parameters will result in a revert
    // The function will only succeed with a newStakingDuration >= 1711843200
    // i.e., if successful, withdrawal will only be possible on June 29th, 2078
    vm.prank(platform);
    staking.updateStakeDuration(1711843200);

    vm.warp(block.timestamp + 1711843200);
    (uint128 stakingEndTimestamp, ) = staking.userStakeMetadata(platform);
    assert(stakingEndTimestamp == block.timestamp); // June 29th, 2078 (3423690000)
}
```

RES-17 Integer Underflow

Launchpad.t.sol (added lines):

```
function testExploitIntegerUnderflow(
    uint128 randomNumOne,
    uint128 randomNumTwo,
    uint128 randomNumThree
) public {
    uint256 IDO_ID = 1;
    prepareLotteryForDefaultIDO(IDO_ID, randomNumOne, randomNumTwo, randomNumThree);

    // Create 90 contestants of which 90 will win
    address[] memory contestants = createContestants(randomNumTwo, 90);
    // All addresses are all duplicates of address(0x1)
    for (uint256 i = 0; i < contestants.length; i++) {
        contestants[i] = address(0x01);
    }

    // Run the lottery
    vm.prank(platform);
    launchpad.runLottery(IDO_ID, Common.Tier.TierTwo, contestants);
}
```